Statistical Learning Theory: A Tutorial

Sanjeev R. Kulkarni and Gilbert Harman*

February 20, 2011

Abstract

In this article, we provide a tutorial overview of some aspects of statistical learning theory, which also goes by other names such as statistical pattern recognition, nonparametric classification and estimation, and supervised learning. We focus on the problem of two-class pattern classification for various reasons. This problem is rich enough to capture many of the interesting aspects that are present in the cases of more than two classes and in the problem of estimation, and many of the results can be extended to these cases. Focusing on two-class pattern classification simplifies our discussion, and yet it is directly applicable to a wide range of practical settings.

We begin with a description of the two class pattern recognition problem. We then discuss various classical and state of the art approaches to this problem, with a focus on fundamental formulations, algorithms, and theoretical results. In particular, we describe nearest neighbor methods, kernel methods, multilayer perceptrons, VC theory, support vector machines, and boosting.

Index Terms: statistical learning, pattern recognition, classification, supervised learning, kernel methods, neural networks, VC dimension, support vector machines, boosting

1 Introduction

In this paper, we focus on the problem of two-class pattern classification — classifying an object into one of two categories based on several observations or measurements of the object. More specifically, we are interested in methods for *learning* rules of classification.

Consider the problem of learning to recognize handwritten characters or faces or other objects from visual data. Or, think about the problem of recognizing spoken words. While

*Sanjeev R. Kulkarni is with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544. His email address is kulkarni@princeton.edu. Gilbert Harman is with the Department of Philosophy, Princeton University, Princeton, NJ 08544. His email address is harman@princeton.edu.

humans are extremely good at these types of classification problems in many natural settings, it is quite difficult to design automated algorithms for these tasks with anywhere near the performance and robustness of humans. Even after more than a half century of effort in fields such as electrical engineering, mathematics, computer science, statistics, philosophy, and cognitive science, humans can still far outperform the best pattern classification algorithms that have been developed. That said, enormous progress has been made in learning theory, algorithms, and applications. Results in this area are deep and practical and are relevant to a range of disciplines.

Our aim in this paper is to provide an accessible introduction to this field, either as a first step for those wishing to pursue the subject in more depth, or for those desiring a broad understanding of the basic ideas. Our treatment is essentially a condensed version of the book [9], which also provides an elementary introduction to the topics discussed here, but in more detail. Although many important aspects of learning are not covered by the model we focus on here, we hope this paper provides a valuable entry point to this area. For further reading, we include a number of references at the end of this paper. The references included are all textbooks or introductory, tutorial, or survey articles. These references cover the topics discussed here as well as many other topics in varying levels of detail and depth. Since this paper is intended as a brief, introductory tutorial, and because all the material presented here is very standard, we refrain from citing original sources, and instead refer the reader to the references provided at the end of this paper and the additional pointers contained therein.

2 The Pattern Recognition Problem

In the pattern classification problem, we observe an object and wish to classify it into one of two classes, which we might call 0 and 1 (or -1 and 1). In order to decide either 0 or 1, we assume we have access to measurements of various properties of the object. These measurements may come from sensors that capture some physical variables of interest, or *features*, of the object.

For simplicity, we represent each measured feature by a single real number. Although in some applications certain features may not be very naturally represented by a number, this assumption allows discussion of the most common learning techniques that are useful in many applications. The set of features can be put together to form a *feature vector*. Suppose there are d features with the value of the features given by $x_1, x_2, \ldots, x_d$. The feature vector is $\bar{x} = (x_1, x_2, \ldots, x_d)$, which can be thought of as a point or a vector in d-dimensional space $\mathbf{R}^d$, which we call *feature space*.

One concrete application is image classification. The features in this case might be the intensities of pixels in the image. For an $N \times N$ gray scale image the total number of features would be $d = N^2$. In the case of a color image, each pixel can be considered as providing 3 measurements, corresponding to the intensities of each of three color components. In this case,

there are $d = 3N^2$ features. For even a modest size image, the dimension of the feature space can be quite large.

In most applications, the class to which an object belongs is not uniquely and definitively determined by its feature vector. There are some fundamental reasons for this. First, although it would be nice if the measured features capture all the properties of the object important for classification, this usually is not the case. Second, depending on the application and the specific measurements, the feature values may be noisy. For these reasons, a statistical formulation for the pattern recognition problem is very useful.

We assume that there are prior probabilities $P(0)$ and $P(1)$ of observing an object from each of the two classes. The feature vector of the object is related to the class to which the object belongs. This is described through conditional probability densities $p(\bar{x}|0)$ and $p(\bar{x}|1)$. Equivalently, let $y \in \{0, 1\}$ be the label associated with $\bar{x}$ (i.e., the actual class to which the object belongs). Then, we can think of the pair $(\bar{x}, y)$ as being drawn from a joint distribution $P(\bar{x}, y)$.

When we observe a feature vector $\bar{x}$, our problem is to decide either 0 or 1. So, a decision rule (or classification rule) can be thought of as a mapping $c : \mathcal{R}^d \to \{0, 1\}$, where $c(\bar{x})$ indicates the decision when feature vector $\bar{x}$ is observed. Equivalently, we can think of a decision rule as a partition of $\mathcal{R}^d$ into two sets Ω_0 and Ω_1 that correspond to those feature vectors that get classified as 0 and 1, respectively. For technical/mathematical reasons we do not consider the set of *all* possible subsets of $\mathcal{R}^d$ as potential decision rules, but rather restrict attention to decision rules that are *measurable*.

Out of the enormous collection of all possible decision rules, we would like to select one that performs well. We make a correct decision if our decision $c(\bar{x})$ is equal to the label y. A natural success criterion is the probability that we make a correct decision. We would like a decision rule that maximizes this probability, or equivalently, that minimizes the probability of error. For a decision rule c, the probability of error of c, denoted $R(c)$, is given by

$$R(c) = \mathrm{E}[(c(\bar{x}) - y)^2] = P(0)P(\Omega_1|0) + P(1)P(\Omega_0|1).$$

In the ideal (and unusual) case, where the underlying probabilistic structure is known, the solution to the classification problem is well known and is a basic result from statistics. On the assumption that the underlying probabilities $P(0)$, $P(1)$ and $p(\bar{x}|0)$, $p(\bar{x}|1)$ are all known, we can compute the so-called posterior probabilities $P(0|\bar{x})$ and $P(1|\bar{x})$ using Bayes theorem. Specifically, the unconditional distribution $P(\bar{x})$ is given by

$$P(\bar{x}) = P(0)P(\bar{x}|0) + P(1)P(\bar{x}|1) \tag{1}$$

and probabilities $P(0|\bar{x})$ and $P(1|\bar{x})$ are then given by

$$P(0|\bar{x}) \quad = \quad \frac{P(0)\,P(\bar{x}|0)}{P(\bar{x})} \tag{2}$$

$$P(1|\bar{x}) \quad = \quad \frac{P(1)\,P(\bar{x}|1)}{P(\bar{x})} \tag{3}$$

Once we have the posterior probabilities, the best decision is simply to choose the class with larger conditional probability – namely, decide 0 if $P(0|\bar{x}) > P(1|\bar{x})$ and decide 1 if $P(1|\bar{x}) > P(0|\bar{x})$. If $P(0|\bar{x}) = P(1|\bar{x})$, then it doesn't matter what we decide. This classification rule is called Bayes decision rule, or sometimes simply Bayes rule.

Bayes decision rule is optimal in the sense that no other decision rule has a smaller probability of error. The error rate of Bayes decision rule, denoted by R^*, is given by

$$R^* = \int_{\mathcal{R}^d} \min\{P(0|\bar{x}), P(1|\bar{x})\} p(\bar{x})\, d\bar{x} \tag{4}$$

Clearly, $0 \leq R^* \leq 1/2$ but we can't say more without additional assumptions.

Finding Bayes decision rule requires knowledge of the underlying distributions, but typically in applications these distributions are not known. In fact, in many applications, even the form of or approximations to the distributions are unknown. In this case, we try to overcome this lack of knowledge by resorting to labeled examples. That is, we assume that we have access to examples $(\bar{x}_1, y_1), \ldots, (\bar{x}_n, y_n)$ that are independent and identically distributed according to the unknown distribution $P(\bar{x}, y)$. Using these examples, we want to come up with a decision rule to classify a new feature vector $\bar{x}$.

The term "supervised" learning arises from the fact that we assume we have access to the training examples $(\bar{x}_1, y_1), \ldots, (\bar{x}_n, y_n)$ that are properly labeled by a "supervisor" or "teacher". This contrasts with "unsupervised learning" in which many examples of objects are available, but the class to which each object belongs is unknown. Other formulations such as semi-supervised learning, reinforcement learning, and related problems have also been widely studied, but in this paper we focus exclusively on the case of supervised learning (and specifically the case of two-class pattern classification). In the following sections, we describe a number of approaches and results for this learning problem.

3 Nearest Neighbor and Kernel Rules

Perhaps the simplest decision rule one might come up with is to find in the training data the feature vector $\bar{x}_i$ that is closest to $\bar{x}$, and then decide that $\bar{x}$ belongs to the same class as given by the label y_i. This decision rule is called the "nearest neighbor rule" (or NN rule, or 1-NN rule).

Associated with each $\bar{x}_i$ is a region (called the Voronoi region) consisting of all points that are closer to $\bar{x}_i$ than to any other $\bar{x}_j$. The NN rule simply classifies a feature vector $\bar{x}$ according

to the label associated to the Voronoi region to which $\bar{x}$ belongs. Figure 1 illustrates Voronoi regions in two dimension. Of course, in general, the feature vectors $\bar{x}_i$ are in a high-dimensional space.

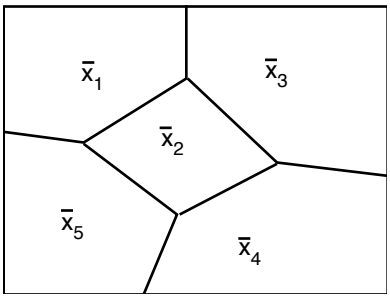

Figure 1: Voronoi Regions

To measure the performance of the NN rule, we consider the expected performance with respect to the new instance to be classified as well as the training examples. Let R_n denote the expected error rate of the NN rule after n training examples, and let R_∞ denote the limit of R_n as n tends to infinity.

It can be shown that

$$R^* \leq R_\infty \leq 2R^*(1 - R^*). \tag{5}$$

A simpler (but looser) bound is

$$R^* \leq R_\infty \leq 2R^*. \tag{6}$$

In general we cannot say anything stronger than the bounds given in the sense that there are underlying probability distributions for which the performance of the NN rule achieves either the upper or lower bound.

Using only random labeled examples but knowing nothing about the underlying distributions, we can (in the limit) achieve an error rate no worse than twice the error rate that could be achieved by knowing everything about the probability distributions. Moreover, we can do this with an extremely simple rule that bases its decision on just the nearest neighbor to the feature vector we wish to classify.

It is natural to ask whether we can do any better by using more neighbors. For example, consider the k-NN rule in which we use the k nearest neighbors, for some fixed number k, and take a majority vote of the labels corresponding to these k nearest neighbors. Let R_∞^k be the error rate of the k-nearest neighbor rule in the limit of infinite data. Under certain conditions, the k-NN rule outperforms the 1-NN rule. However, it can also be shown that there are some distributions for which the 1-NN rule outperforms the k-NN rule for any fixed $k > 1$.

A very useful idea is to let the number of neighbors used grow with n (the amount of data we have). That is, we can let k be a function of n, so that we use a k_n-NN rule. We need

$k_n \to \infty$ so that we use more and more neighbors as the amount of training data increases. But we should make sure that $k_n/n \to 0$, so that asymptotically the number of neighbors we use is a negligible fraction of the total amount of data. This will ensure that we use neighbors that get closer and closer to the observed feature vector $\bar{x}$. For example, we might let $k_n = \sqrt{n}$ to satisfy both conditions.

It turns out that with any such k_n (such that $k_n \to \infty$ and $k_n/n \to 0$ are satisfied), we get $R_n^{k_n} \to R^*$ as $n \to \infty$. That is, in the limit as the amount of training data grows, the performance of the k_n-NN rule approaches that of the optimal Bayes decision rule.

What is surprising about this result is that by observing data but without knowing anything about the underlying distributions, asymptotically we can do as well as if we knew the underlying distributions completely. And, this works without assuming the underlying distributions take on any particular form or satisfy any stringent conditions. In this sense, the k_n-NN rule is called *universally consistent*, and is an example of truly *nonparametric* learning in that the underlying distributions can be arbitrary and we need no knowledge of their form. It was not known until the early 1970's whether universally consistent rules existed, and it was quite surprising when the k_n-NN rule along with some others were shown to be universally consistent. A number of such decision rules are known today, as we will discuss further in subsequent sections.

However, universal consistency is not the end of the story. A critical issue is that of convergence rates. Many results are known on the convergence rates of the nearest neighbor rule and other rules. A fairly generic problem is that, except under rather stringent conditions, the rate of convergence for most methods is very slow in high dimensional spaces. This is a facet of the so-called "curse of dimensionality." In many real applications the dimension can be extremely large, which bodes ill for many methods. Furthermore, it can be shown that there are no "universal" rates of convergence. That is, for any method, one can always find distributions for which the convergence rate is arbitrarily slow. This is related to so-called "No Free Lunch Theorems" that formalize the statement that there is no one method that can universally beat out all other methods. These results make the field continue to be exciting, and makes the design of good learning algorithms and the understanding of their performance an important science and art. In the following sections, we will discuss some other methods useful in practice.

4 Kernel Rules

Rather than fixing the number of neighbors, we might consider fixing a distance h and taking a majority vote of the labels y_i corresponding to all examples from among $\bar{x}_1, \ldots, \bar{x}_n$ that fall within a distance h of $\bar{x}$. If none of the $\bar{x}_i$ fall within distance h or if there is a tie in the majority vote, we need some way to decide in these cases. One way to do this is simply to break ties randomly.

As with nearest neighbor rules, this rule classifies a feature vector $\bar{x} \in \mathbf{R}^d$ according to a majority vote among the labels of the training points $\bar{x}_i$ in the vicinity of $\bar{x}$. However, while the nearest neighbor rule classifies $\bar{x}$ based on a specified number k_n of training examples that are closest to $\bar{x}$, this rule considers all $\bar{x}_i$'s that are within a fixed distance h of $\bar{x}$. The rule we have been discussing is the simplest example of a very general class of rules call kernel rules.

One way to precisely describe this rule is using a "kernel" (or "window function") defined as follows:

$$K(\bar{x}) = \begin{cases} 1 & \text{if } \|\bar{x}\| \leq 1 \\ 0 & \text{otherwise,} \end{cases} \tag{7}$$

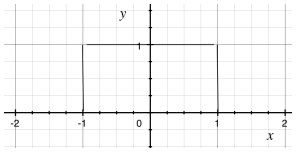

Figure 2: Basic Window Kernel

Define vote counts $v_n^0(\bar{x})$ and $v_n^1(\bar{x})$ as

$$v_n^0(\bar{x}) = \sum_{i=1}^{n} I_{\{y_i=0\}} K\left(\frac{\bar{x} - \bar{x}_i}{h}\right)$$

and

$$v_n^1(\bar{x}) = \sum_{i=1}^{n} I_{\{y_i=1\}} K\left(\frac{\bar{x} - \bar{x}_i}{h}\right).$$

The decision rule is then to decide class 0 if $v_n^0(\bar{x}) > v_n^1(\bar{x})$, decide class 1 if $v_n^1(\bar{x}) > v_n^0(\bar{x})$, and break ties randomly.

Writing the rule in this way naturally suggests that we might pick a different kernel function $K(\cdot)$. For example, it makes sense that training examples very close to $\bar{x}$ should have more influence (or a larger weight) in determining the classification of $\bar{x}$ than those which are farther away. Such a function is usually nonnegative and is often monotonically decreasing along rays starting from the origin. In addition to the window kernel, some other popular choices for kernel functions include:

Triangular kernel:	$K(\bar{x}) = (1 - \|\bar{x}\|) I_{\{\|\bar{x}\| \leq 1\}}$
Gaussian kernel:	$K(\bar{x}) = e^{-\|\bar{x}\|^2}$.
Cauchy kernel:	$K(\bar{x}) = 1/(1 + \|\bar{x}\|^{d+1})$
Epanechnikov kernel:	$K(\bar{x}) = (1 - \|\bar{x}\|^2) I_{\{\|\bar{x}\| \leq 1\}}$.

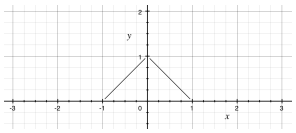

Figure 3: Triangular Kernel

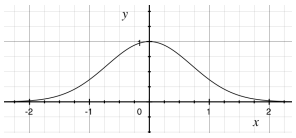

Figure 4: Gaussian Kernel

The positive number h is called the *smoothing factor*, or *bandwidth*, and is an important parameter of a kernel rule. If h is small, the rule gives large relative weight to points near $\bar{x}$, and the decision is very "local," while for a large h many more points are considered with fairly large weight, but these points can be farther from $\bar{x}$. Hence, h determines the amount of "smoothing." In choosing a value for h, one confronts a similar kind of tradeoff as in choosing the number of neighbors using a nearest neighbor rule.

As we might expect, to get universal consistency, we need to let the smoothing factor depend on the amount of data, so we let $h = h_n$. To make sure we get "locality" (i.e., so that the training examples used get closer to $\bar{x}$), we need to have $\lim_{n\to\infty} h_n = 0$. To make sure that the number of training examples used grows, we need to have $\lim_{n\to\infty} nh_n^d = \infty$.

These two conditions ($h_n \to 0$ and $nh_n^d \to \infty$) are analogous to the conditions imposed on k_n to get universal consistency. In addition to these two conditions, to show universal consistency we need certain fairly mild regularity conditions on the kernel function $K(\cdot)$. In particular, we need $K(\cdot)$ to be non-negative, and over a small neighborhood of the origin $K(\cdot)$ is required to be larger than some fixed positive number $b > 0$. The last requirement is more technical but as a special case, it's enough if we require that $K(\cdot)$ be non-increasing with distance from the origin and have finite volume under the kernel function.

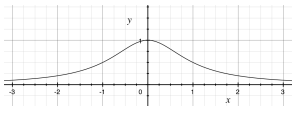

Figure 5: Cauchy Kernel

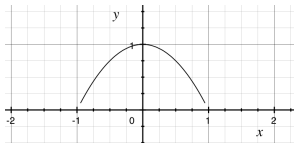

Figure 6: Epanechnikov Kernel

5 Multilayer Perceptrons

Neural networks (or, neural nets, artificial neural networks) are collections of (usually simple) processing units each of which is connected to many other units. With tens or hundreds of units and several times that many connections, a network rapidly can get complicated. Understanding the behavior of even small networks can be difficult, especially if there can be "feedback" loops in the connectivity structure of the network (i.e., the output of a neuron can serve as inputs to others which via still others come back and serve as inputs to the original neuron).

We'll focus on a special class of networks called multilayer perceptrons. The units are organized in a multilayer feedforward network (Figure 7).

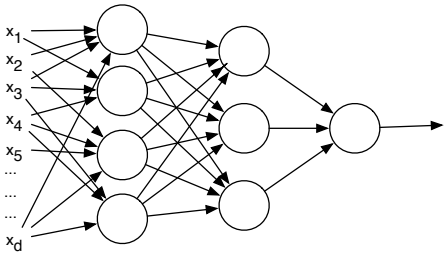

Figure 7: Feed Forward Network

That is, the units are organized in layers, with the output of neurons in one layer serving as the inputs to the neurons in the next layer. Because there are no feedback loops, the behavior of the network is simple to compute.

The first layer receives external inputs, which will be features of the objects we wish to classify. The output of each unit in one layer is passed as an input to the units in the next layer. The outputs of the last layer (called the output layer), which gives the final result computed by the network for the given input.

The particular classification rule implemented by a network is determined by the specifics of the network architecture and the computation done by each neuron. A crucial part of the network computation is a set of parameters called weights. There is usually a real-valued

weight associated with each connection between units. The weights are generally considered to be adjustable, while the rest of the network is usually thought of as fixed. Thus, we think of the classification rule implemented by the network as being determined by the weights, and this classification rule can be altered if we change the weights.

To solve a given pattern recognition problem with a neural net, we need a set of weights that results in a good classification rule. As we discussed previously, it is difficult to directly specify good decision rules in most practical situations, and this is made even more difficult by the complexity of the computation performed by the neural net. So, how should we go about selecting the weights of the network?

This is where learning comes in. Suppose we start with some initial set of weights. It is unlikely that the weights we start with will result in a very good classification rule. But, as before, we will assume we have a set of labeled examples (training data). If we use this data to "train" the network to perform well on this data, then perhaps the network will also "generalize" and provide a decision rule that works well on new data. The success of neural networks in many practical problems is due to an efficient way to train a given network to perform well on a set of training examples, together with the fact that in many cases the resulting decision rule does in fact generalize well.

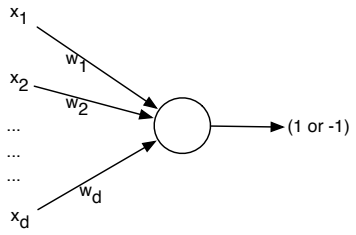

Figure 8: Perceptron

In a multilayer perceptron, each single unit is called a perceptron (depicted in Figure 8) that operates as follows. The feature vector $x_1, x_2, \ldots, x_d$ is presented as the input so that input x_i is connected to the unit with an associated weight w_i.

The output a of the perceptron is given by

$$a = \text{sign}(x_1 w_1 + \cdots + x_d w_d)$$

where

$$\text{sign}(u) = \begin{cases} -1 & \text{if } u < 0 \\ 1 & \text{otherwise} \end{cases}$$

In general, a perceptron can have a non-zero threshold but often for convenience a threshold

of 0 is assumed, as we will do here. This is not restrictive since a non-zero threshold can be mimicked by a unit with a zero threshold and an extra input.

A simple learning rule, called the Perceptron Convergence Procedure, can be used for perceptrons to classify the well data as the perceptron can. The problem is that a perceptron can represent only linear decision rules (i.e., those with a hyperplane decision boundary).

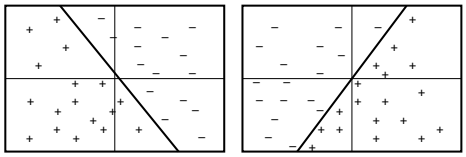

Figure 9: Hyperplane decision rules. In $\mathbf{R}^2$ these are just straight lines.

Linear classifiers have been extremely well-studied, and in some applications, the best linear decision rule may be good enough (or even optimal). But, in other problems we may need more general classifiers. There may be no reason to believe that the optimal Bayes decision rule (or even just a reasonably good rule) is linear.

Multilayer networks overcome the representation limitations of perceptrons. Perhaps surprisingly, with just three layers and enough units in each layer, a multilayer network can approximate any decision rule. In fact, the third layer (the output layer) consists of just a single output unit, so multiple units are needed only in the first and second layer.

However, due to the complexity of the network and the nonlinear relationships between the weights and the network output, a learning method for finding good weights is more difficult.

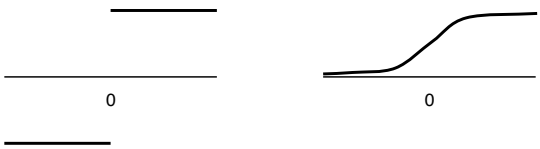

Figure 10: Threshold functions versus sigmoid functions

Because of this, it is very common in multilayer perceptrons to replace the threshold output function of the perceptron with a "smoothed" version such as that shown on the right in Figure 10. It turns out that having the smooth function vary between 0 and 1 instead of -1 and 1 simplifies things. So, for convenience, we make this change as well. A function with this general "S" shape that is differentiable, increasing, and tends to finite limits at $-\infty$ and ∞ is called a *sigmoid function*.

Thus, the output of a unit is now given by

$$a = \sigma(x_1 w_1 + \cdots + x_d w_d) \tag{8}$$

where the x_i are the inputs to the unit, the w_i are the weights, and $\sigma(\cdot)$ is a sigmoid function. One commonly used sigmoid function is

$$\sigma(y) = \frac{1}{1 + e^{-y}} \tag{9}$$

With a sigmoidal function small changes in one of the weights or inputs to the unit will result in small changes to the output. This eliminates the discontinuity problem associated with the threshold function and allows some basic but powerful machinery from calculus and optimization to be used to deal with the learning problem of adjusting the weights in the network to perform well on the training data.

The most common learning rule for multilayer perceptrons is called backpropagation. This is essentially a gradient descent type of procedure that sequentially cycles through the training examples and incrementally adjusts the weights in a way to as to try to reduce the error between the training examples and the network output. Due to limited space, we do not provide a detailed explanation of backpropagation here, but this can be found in a number of books on statistical pattern recognition, learning theory, and neural networks given in the bibliography. Like all "descent" optimization algorithms, backpropagation finds only local minima. Nevertheless, it has been shown to give good results in practice, and backpropagation or its variants are the most widely used training algorithms for multilayer perceptrons.

6 Vapnik Chervonenkis Theory

Neural nets are a special case of a more general approach that fixes a collection of rules C and attempts to find a good rule from this fixed collection. In addition to the practical reasons for working with a fixed collection of decision rules, this perspective also leads to a fruitful conceptual framework. The results obtained provide a characterization of the difficulty of a learning problem in terms of a measure of the "richness" of the class of decision rules.

Once the class of rules is fixed, we would like some measure of the difficulty of learning in terms of the collection C. Rather than saying something about a particular method for selecting a rule from the class, we would like to say something about the inherent difficulty due to the richness of the class itself. For example, for neural nets, we would like to make statements about the fundamental limits of the network — not just statements about the performance of back-propagation in particular. We would also like some measure of how much training data is required for learning. Vapnik-Chervonenkis theory provides results of this sort.

If there is only one possible decision rule in C, then "learning" is a non-issue. There is no choice but to select this one decision rule, regardless of any data observed. The issue of learning arises when C contains many rules and we need to select a good rule from C based on the training examples.

Each rule $c \in C$ has an associated error rate $R(c)$. A natural quantity to consider is

$$R_C^* = \min_{c \in C} R(c)$$

which is the best performance we can hope for if we are restricted to using rules from C. Of course, we always have $R_C^* \geq R^*$. If C is rich, then R_C^* may be close to the Bayes error rate R^*, but in other cases R_C^* may be far from R^*.

Note that we cannot actually compute the error rates for the rules in C, since we do not know the necessary probability distributions. Instead, we have to try to select a good rule from C based on the data.

With a finite amount of data, it may be unreasonable to expect to always be able to find the *best* rule from C. We will settle for a rule that is only approximately optimal (or *approximately correct*). That is, we would like to select a hypothesis h from C such that the error rate $R(h)$ of the hypothesis satisfies $R(h) \leq R_C^* + \epsilon$, where ϵ is an accuracy parameter.

Moreover, since the training examples are random, we will require only that we produce a good hypothesis with high probability. In other words, we will require that

$$P\{R(h) \leq R_C^* + \epsilon\} \geq 1 - \delta$$

for some confidence parameter δ. This is the *Probably Approximately Correct* (or PAC) criterion.

Of course, we should expect that the number of examples we need to learn will depend on the choice of ϵ and δ. However, we require that the number of training examples required by a learning algorithm not depend on the underlying distributions.

Specifically, we say that a class of decision rules C is PAC learnable if there is a mapping that produces a hypotheses $h \in C$ based on training examples such that for every $\epsilon, \delta > 0$ there is a finite sample size $m(\epsilon, \delta)$ such that for any distributions we have

$$P\{R(h) > R_C^* + \epsilon\} < \delta$$

after seeing $m(\epsilon, \delta)$ training examples.

There is an inherent trade-off involved in the richness of this class C. If C is too rich, then it may be easy to find a rule that fits the data, but we won't have confidence that performance on the training data will be predictive of performance on new data. On the other other hand, if C is not very rich, then perhaps no rule from C will have very good performance (even if we can

be confident of finding the best rule from C). Simply counting the number of rules in C is not the right measure of complexity (or richness) of C. Instead a measure of richness should take into account the "expressive power" of rules in C, and this is what the notion of "shattering" and VC dimension capture.

Given a set of feature vectors $\bar{x}_1, \ldots, \bar{x}_n$, we say that $\bar{x}_1, \ldots, \bar{x}_n$ are *shattered* by a class of decision rules C if all 2^n labelings of the feature vectors $\bar{x}_1, \ldots, \bar{x}_n$ can be generated using rules from C.

The *Vapnik-Chervonenkis dimension* (or *VC dimension*) of a class of decision rules C, denoted VCdim(C), is the largest integer V such that *some* set of V feature vectors is shattered by C. If arbitrarily large sets can be shattered, then VCdim(C) $= \infty$.

Under certain mild measurability conditions, it can be shown that C is PAC learnable iff the VC dimension of C is finite. Moreover, if $V =$ VCdim(C) satisfies $1 \leq V < \infty$ then a sample size

$$\frac{64}{\epsilon^2}\left(2V \log\left(\frac{12}{\epsilon}\right) + \log\left(\frac{4}{\delta}\right)\right)$$

is sufficient for ϵ, δ learning. What is more important than the exact constants is the form of the bound and the fact that such a precise statement can be made. Lower bounds can also be obtained which state that ϵ, δ learning is not possible unless a certain number of examples are used, and the lower bounds have similar dependence on ϵ, δ, and V.

These results can be applied to neural networks as follows. Recall that the set of decision rules that can be represented by a single perceptron is the set of halfspaces. If the threshold is 0 the halfspaces pass through the origin, but for adjustable thresholds we get all halfspaces. It can be shown that in 2 dimensions, the set of halfspaces has VC dimension equal to 3. More generally, the set of all halfspaces in d dimensions can be shown to have VC dimension equal to $d + 1$. Thus, the learning results can be directly applied to learning using perceptrons.

Results have also been obtained for multilayer networks. Although it is quite difficult to compute the VC dimension exactly, useful bounds have been obtained. For networks with threshold units, one bound is that

$$\text{VCdim}(C) \leq 2(d+1)(s)\log(es)$$

where d is the dimension of the feature space, s is the total number of perceptrons, and e is the base of the natural logarithm (approximately 2.718).

Similar results have been obtained for the case of sigmoidal units. In this case, the bound also involves the maximum slope of the sigmoidal output function and the maximum allowed magnitude for the weights.

The results involving finite VC dimension focus on the *estimation* error, the problem of trying to predict which rule from C will be the best based on a set of random examples. If

the class of rules is too rich, then even with a large number of examples it can be difficult to distinguish those rules from C that will actually perform well on new data from those that just happen to fit the data seen so far but have no predictive value.

In addition to the estimation error, the *approximation* error is also important in determining the overall performance of a decision rule. The approximation error sets a limit on how well we can do no matter how much data we receive. If we are restricted to using rules from the class C, we can certainly do no better than the best rule from C. Once we fix a class with finite VC dimension, we are stuck with whatever approximation error results from the class and the underlying distributions. Moreover, since the distributions are unknown, the actual approximation error we must live with is also unknown.

If we slightly modify the PAC criterion, we can deal with certain classes that have infinite VC dimension, and thereby address the issue of non-zero approximation errors. Consider a sequence of classes $C_1, C_2, \ldots$ that are nested, so that $C_1 \subset C_2 \subset \cdots$. If $V_i = \text{VCdim}(C_i)$, then we require $V_i < \infty$ for all i, but we allow $V_i \to \infty$ as $i \to \infty$.

Given such a sequence of classes, we can think of $\text{VCdim}(C_i)$ as a measure of the "complexity" of the *class* of rules C_i. We can identify the complexity of a *rule* with the smallest index of a class C_i to which the rule belongs. The collection of all rules under consideration is given by

$$C = \bigcup_{i=1}^{\infty} C_i.$$

If $V_i \to \infty$ then the class C has infinite VC dimension and so is not PAC learnable.

Nevertheless, it may turn out that we can still find good rules from C as we get more and more data. The key idea is to allow the number of examples needed for ϵ, δ learning to depend on the underlying probability distributions (as well as on ϵ and δ), which is both an intuitive and reasonable idea. We should expect that complicated or difficult problems will require more data for learning, while simple problems can be learned easily with minimal data.

With this modified criterion, a hierarchy of classes $C = \cup_{i=1}^{\infty} C_i$ can be learned by trading off the fit to the data against the complexity of the hypothesis. This idea is at the heart of various techniques that go by different names: Occam's razor, minimum description length (MDL) principle, structural risk minimization, Akaike's information criterion (AIC), etc. Our choice of hypothesis should reflect some tradeoff between misfit on the data (error) and complexity. Given specific ways of measuring the error of a decision rule on the data and the complexity of a decision rule, we can select a hypothesis by

$$h = \text{argmin}_{h \in C} \; [\text{error}(h) + \text{complexity}(h)]$$

Making the first term small favors choosing a hypothesis in some C_i with large i. However, if unchecked this would result in "overfitting" and wouldn't provide much predictive power. The

second term helps to control the "overfitting" problem. By striking a balance between these two terms, we control the expressive power of the rules we will entertain, and also consider how well the rules take into account the training data.

There are many choices for exactly how to carry out the misfit versus complexity tradeoff. One concrete result is as follows. Suppose the hierarchy of classes is chosen so that $R^*_{C_i} \to R^*$ as $i \to \infty$. This means that we can find rules with performance arbitrarily close to the optimal Bayes decision rule as i gets sufficiently large. We select a sequence $k_n \to \infty$ such that

$$\frac{V_{k_n} \log(n)}{n} \to 0 \text{ as } n \to \infty \tag{10}$$

After seeing n examples, we select a rule h_n from the class C_{k_n} that fits the data the best. Then it can be shown that this method is *universally consistent*. That is, for any distributions, as $n \to \infty$ we have $R(h_n) \to R^*$. Hence, as we get more and more data, the performance of the rule we select gets closer and closer to the Bayes error rate.

However, in general we cannot obtain uniform sample size bounds that will be guarantee ϵ, δ learning after some fixed number of examples. The number of examples needed will depend on the underlying distributions, and these are unknown. The performance will also depend on the choice of hierarchy of classes and the misfit/complexity tradeoff. These choices are often a matter of art, intuition, and technical understanding of the problem domain. The complexity hierarchy of the decision rules reflects the learner's inductive bias about which rules are considered "simple", and how well this reflects reality has a strong effect on performance.

7 Support Vector Machines

Support Vector Machines (SVMs) provide a state-of-the-art learning method that has been highly successful in a variety of applications. SVMs are a special case of generalized kernel methods, a special case of which were described in Section 4.

The origin of SVMs arises from two key ideas. The first idea is to map the feature vectors in a nonlinear way to a high (possibly infinite) dimensional space and then utilize linear classifiers in this new space. Specifically, let $\mathcal{H}$ denote the new feature space, and let Φ denote the mapping, so that

$$\Phi : \mathbf{R}^d \to \mathcal{H}$$

For an original feature vector $\bar{x} \in \mathcal{R}^d$, the transformed feature vector is given by $\Phi(\bar{x})$. The label y remains the same. Thus, the training example $(\bar{x}_i, y_i)$ becomes $(\Phi(\bar{x}_i), y_i)$.

Then, we seek a hyperplane in the transformed space $\mathcal{H}$ that separates the transformed training examples $(\Phi(\bar{x}_1), y_1), \ldots, (\Phi(\bar{x}_n), y_n)$. That is, we want to find a hyperplane in the space $\mathcal{H}$ so that a transformed feature vector $\Phi(\bar{x}_i)$ lies on one side of the hyperplane if the

label $y_i = -1$, and $\Phi(\bar{x}_i)$ lies on the other side of the hyperplane if $y_i = 1$. As in in Section 5, it is convenient here to assume the class labels are -1 and 1 instead of 0 and 1.

This results in nonlinear classifiers in the original space, which overcomes the representational limitations of linear classifiers. However, the use of linear classifiers (in the transformed space) lends itself to computational methods for finding a classifier that performs well on the training data. These computational benefits are retained while allowing a rich class of nonlinear rules.

The second key idea is to try to find hyperplanes that separate the data with a *large margin*, that is a hyperplane separates the data as much as possible from among the generally infinitely many hyperplanes that may separate the data. While many separating hyperplanes perform equally well on the training data (in fact perfectly well if they separate the data), the generalization performance on new data can vary significantly.

Figure 11 shows a separable case in two dimensions with a number of separating lines. All of these separating hyperplanes perform equally well on the training data (in fact they perform perfectly), but they might have different generalization performance. It is natural to ask whether some of the separating hyperplanes are better than others in terms of their error rate on new examples. Without some qualification, the answer is no. In a worst case sense, all hyperplanes will result in similar performance. However, it may be that the distributions giving rise to the worst-case performance are unusual, and that for most "typical" distributions we might be able to do better by exploiting some properties of the training data.

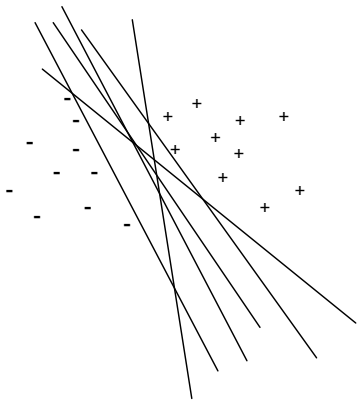

Figure 11: Separating Hyperlanes in Two Dimensions

It turns out that finding a hyperplane with large margin helps to provide good generalization performance. Intuitively, if the margin is large then the separation of the training examples is robust to small changes in the hyperplane, and we would expect the classifier to have better predictive performance. This is often the case, though not in a worst-case sense. The fact that large margin classifiers tend to have good generalization performance has been both justified

theoretically and observed in practice.

Let d_+ denote the smallest distance from examples labeled 1 to the separating hyperplane, and let d_- denote the smallest distance from examples labeled -1 to the separating hyperplane. The margin of the hyperplane is defined to be $d_+ + d_-$. By choosing the right orientation of a separating hyperplane, we can make $d_+ + d_-$ as large as possible. Any plane parallel to this will have the same value of $d_+ + d_-$.

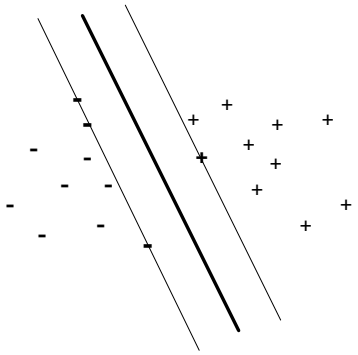

Figure 12: Large Margin Separation

In Figure 12, we show this in two dimensions. The hyperplane for which $d_+ = d_-$ is shown, together with parallel hyperplanes for which either $d_+ = 0$ or $d_- = 0$. These last two hyperplanes will pass through one or more examples from the training data. These examples are called the *support vectors*, and they are the examples that define the maximum margin hyperplane. If we move or remove the support vectors, then the maximum margin hyperplane can change, but if we move or remove any of the other training examples, then the maximum margin hyperplane remains unchanged.

Recall that the transformed training examples are $(\Phi(\bar{x}_1), y_1), \ldots, (\Phi(\bar{x}_n), y_n)$ where $\Phi(\bar{x}_i) \in \mathcal{H}$ and $y_i \in \{-1, 1\}$. The equation of a hyperplane in $\mathcal{H}$ can be represented in terms of a vector $\bar{w}$ and a scalar b as

$$\bar{w} \cdot \Phi(\bar{x}) + b = 0$$

It turns out that $\bar{w}$ is the normal to the hyperplane and $|b|/\|\bar{w}\|$ is the distance of the hyperplane from the origin.

Since there are finitely many training examples, if a hyperplane separates the training data, then each training example must be at least β away from the hyperplane for some $\beta > 0$. Then we can then can renormalize to require that a separating hyperplane satisfies

$$\begin{aligned} \Phi(\bar{x}_i) \cdot \bar{w} + b &\geq +1 \quad \text{if } y_i = +1 \\ \Phi(\bar{x}_i) \cdot \bar{w} + b &\leq -1 \quad \text{if } y_i = -1 \end{aligned}$$

Equivalently, we can write these conditions as

$$y_i(\Phi(\bar{x}_i) \cdot \bar{w} + b) - 1 \geq 0 \quad \text{for} \quad i = 1, \ldots, n \tag{11}$$

In general, we may not be able to separate the training data with a hyperplane. However, we can still seek a hyperplane that separates the data "as much as possible" while also trying to maximize the margin, with these objectives suitably defined. In order to carry this out, we introduce "slack variables" ξ_i with $\xi_i \geq 0$ for $i = 1, \ldots, n$ and try to satisfy

$$\begin{aligned} \Phi(\bar{x}_i) \cdot \bar{w} + b &\geq +1 - \xi_i \quad \text{if } y_i = +1 \\ \Phi(\bar{x}_i) \cdot \bar{w} + b &\leq -1 + \xi_i \quad \text{if } y_i = -1 \end{aligned}$$

Without a constraint on the ξ_i, the conditions above can be satisfied trivially. We can get a useful formulation by adding a penalty term of the form $C \sum_i \xi_i$, where C is some appropriate constant.

Also, written as in Equation (11), $d_+ = d_- = 1/\|\bar{w}\|$ so that

$$\text{margin} = d_+ + d_- = \frac{2}{\|\bar{w}\|}$$

To maximize the margin we can minimize $\|\bar{w}\|$, or equivalently minimize $\|\bar{w}\|^2$.

Thus, we seek a hyperplane that solves the following optimization problem:

$$\begin{aligned} \text{minimize} \quad & \|\bar{w}\|^2 + C \textstyle\sum_i \xi_i \\ \text{subject to} \quad & y_i(\Phi(\bar{x}_i) \cdot \bar{w} + b) - 1 + \xi_i \geq 0 \quad \text{for } i = 1, \ldots, n \\ & \xi_i \geq 0 \quad \text{for } i = 1, \ldots, n \end{aligned}$$

Using techniques from optimization (Lagrange multipliers and considering the dual problem), the maximum margin separating hyperplane can be found by solving the following optimization problem:

$$\begin{aligned} \text{maximize} \quad & \textstyle\sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j \, y_i y_j \, (\Phi(\bar{x}_i) \cdot \Phi(\bar{x}_j)) \\ \text{subject to} \quad & \textstyle\sum_i \alpha_i y_i = 0 \\ & \alpha_i \geq 0 \quad \text{for} \quad i = 1, \ldots, n \end{aligned}$$

This is a standard type of optimization problem called a convex quadratic programming problem for which there are well-known and efficient algorithms. Through solving this optimization, the following equations are satisfied, which provide the equation for the hyperplane:

$$\bar{w} = \sum_{i=1}^{n} \alpha_i y_i \Phi(\bar{x}_i) \tag{12}$$

$$\alpha_i(y_i(\bar{w} \cdot \Phi(\bar{x}_i) + b) - 1) = 0 \text{ for } i = 1, \ldots, n \tag{13}$$

$$y_i(\Phi(\bar{x}_i) \cdot \bar{w} + b) - 1 \geq 0 \text{ for } i = 1, \ldots, n \tag{14}$$

The solution to the optimization problem returns values for the α_i and the ξ_i. From these, $\bar{w}$ can be obtained directly from the expression above in Equation (12). The scalar b can be obtained by solving Equation (13) for any i in terms of b, though a better approach is to do this for all i and take the average of the values obtained as the choice for b.

Solving this optimization problem and obtaining the hyperplane (namely $\bar{w}$ and b) is the process of training. For classification, we need only check on which side of the hyperplane a feature vector $\bar{x}$ falls. That is, $\bar{x}$ is classified as 1 if

$$\bar{w} \cdot \Phi(\bar{x}) + b = \sum_{i=1}^{n} \alpha_i y_i \left(\Phi(\bar{x}_i) \cdot \Phi(\bar{x})\right) + b > 0$$

and $\bar{x}$ is classified as -1 otherwise.

For some i, it turns out that $\alpha_i = 0$. For these i, the corresponding example $(\Phi(\bar{x}_i), y_i)$ does not affect the maximum margin hyperplane. For other i, we have $\alpha_i > 0$, and the examples corresponding to these i do affect the maximum margin hyperplane. These examples (corresponding to positive α_i) are the support vectors.

An important practical consideration is how to implement the transformation $\Phi(\cdot)$. The original feature vector $\bar{x}$ is often high-dimensional itself and the transformed space is typically of even much higher dimension, possibly even infinite dimensional. Thus, computing $\Phi(\bar{x}_i)$ and $\Phi(\bar{x})$ can be very difficult.

A useful result allows us to address this issue and also provides a connection between SVMs and kernel methods discussed in Section 4. Under certain conditions the dot product $\Phi(\bar{x}_i) \cdot \Phi(\bar{x})$ can be replaced with a function $K(\bar{x}_i, \bar{x})$ that is easy to compute. This function $K(\cdot, \cdot)$ is just a kernel function, and the resulting classifier becomes a form of a general kernel classifier. Namely, a feature vector $\bar{x}$ is classified according to whether

$$\begin{aligned} \bar{w} \cdot \Phi(\bar{x}) + b &= \sum_{i=1}^{n} \alpha_i y_i \left(\Phi(\bar{x}_i) \cdot \Phi(\bar{x})\right) + b \\ &= \sum_{i=1}^{n} \alpha_i y_i \, K(\bar{x}_i, \bar{x}) + b \end{aligned}$$

is greater than zero or less than zero.

The terms in the optimization problem used to find the α_i also contain dot products of the form $\Phi(\bar{x}_i) \cdot \Phi(\bar{x}_j)$ which can be replaced with $K(\bar{x}_i, \bar{x}_j)$.

In practice, one generally directly chooses the kernel function K, while the mapping $\Phi(\cdot)$ and the transformed space $\mathcal{H}$ are induced by the choice of K. In fact, once we specify a kernel, the training and classification rule can be implemented directly without the need for even knowing what are the corresponding Φ and $\mathcal{H}$. It turns out that for a given choice of the kernel K,

corresponding Φ and $\mathcal{H}$ exist if and only if K satisfies a condition known as Mercer's condition — i.e., for all $g(\bar{x})$ such that $\int g(\bar{x})^2\, d\bar{x} < \infty$, we have $\int K(\bar{x}, \bar{z}) g(\bar{x}) g(\bar{z})\, d\bar{x}\, d\bar{z} \geq 0$.

Some common choices of kernel functions are those mentioned in Section 4. As with kernel methods in general, the choice of the kernel function and associated parameters is an art and can have a strong influence on classification performance.

8 Boosting

Boosting is an iterative procedure for improving the performance of any learning algorithm. It is among the most successful learning methods available.

Boosting combines a set of "weak" classification rules to produce a "strong" composite classifier. A weak learning rule is one that performs strictly better than random guessing. That is, a weak learning rule and has an error rate $\epsilon = 1/2 - \gamma$ for some $\gamma > 0$. Because of computational or other considerations, it might be easy to produce a weak learning rule, even though finding a good rule may be difficult. This is where boosting comes into play. If we have an algorithm that given training data can find a weak classifier, then boosting can be used to produce a new classifier with much better performance.

Boosting proceeds in a series of rounds. In each round, a weak rule is produced by running some basic learning algorithm using a different weighting of the training examples. Starting with equal weighting in the first round, the weighting of the training examples is updated after each round to place more weight on those training examples that are misclassified by the current weak hypothesis just produced and less weight on those that are correctly classified. This forces the weak learner to concentrate in the next round on these hard-to-classify examples.

Specifically, let $D_t(\cdot)$ denote the distribution on the training examples at round t, so that $D_t(i)$ is the probability (or weight) that we assign to the i-th training example. At stage t, the goal of the weak learning algorithm will be produce a hypothesis $h_t(\cdot)$ that attempts to minimize the weighted error, ϵ_t defined by

$$\epsilon_t = \sum_{i=1}^{n} D_t(i) I_{\{h_t(\bar{x}_i) \neq y_i\}} \tag{15}$$

ϵ_t as the probability of error of the classifier $h_t(\cdot)$, where the probability is computed with respect to the distribution $D_t(\cdot)$. To compute h_t, "new" examples can be generated by drawing examples from the training data $(\bar{x}_i, y_i)$ according to these probabilities, and then train the learning algorithm on this set of "new" examples. Or, if possible, we can simply apply the learning algorithm with the objective of minimizing the weighted training error.

After a number of rounds, a final classification rule is produced via a weighted sum of

the weak rules. Given a set of classification rules $h_1(\bar{x}), h_2(\bar{x}), \ldots, h_T(\bar{x})$, we can form a new classification rule as a weighted combination of these as follows. Define

$$H(\bar{x}) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(\bar{x})\right)$$

where sign(u) returns -1 if $u < 0$ and returns 1 otherwise. For a given $\bar{x}$, the weights α_t are used to combine the decisions of the individual rules $h_t(\bar{x})$, and H outputs either $+1$ or -1 depending on the sign of this weighted sum.

There are a number of variants of the basic boosting algorithm, one of the most popular of which is the AdaBoost algorithm developed by Freund and Schapire (1995). Adaboost has been found to be an extremely effective algorithm and has been widely studied. It works as follows.

- **Input:**
  - Training data $(\bar{x}_1, y_1), \ldots, (\bar{x}_n, y_n)$.
  - A weak learning algorithm.
- **Initialization:**
  - Set $t = 1$.
  - Set $D_1(i) = 1/n$.
- **Main Procedure:** For $t = 1, \ldots, T$:
  - Use the weak learning algorithm on distribution D_t to get classifier $h_t(\cdot)$.
  - Let ϵ_t be the error rate of $h_t(\cdot)$ with respect to the distribution D_t. Set $\alpha_t = \frac{1}{2}\log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$.
  - Update the distribution as follows:

    $$D_{t+1}(i) = \frac{D_t(i)e^{-\alpha_t y_i h_t(\bar{x}_i)}}{Z_t}$$

    where Z_t is a normalization factor to ensure that

    $$\sum_{t=1}^{T} D_{t+1}(i) = 1$$

- **Output:** The final classifier $H(\cdot)$ output by the boosting procedure is given by

$$H(\bar{x}) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(\bar{x})\right)$$

To discuss the performance of Adaboost, let $\gamma_t = 1/2 - \epsilon_t$. Since $\epsilon_t < 1/2$, $\gamma_t > 0$ and is the amount by which h_t performs better than random guessing on the training data (weighted according to D_t).

It can be shown that the training error of the final classifier H produced by AdaBoost is bounded as follows:

$$\frac{1}{n}\sum_{i=1}^{n} I_{\{H(\bar{x}_i)\neq y_i\}} \leq e^{-2\sum_{t=1}^{T}\gamma_t^2}. \tag{16}$$

The bound on the right hand side of Equation (16) gets smaller with every round, and if $\gamma_t \geq \gamma$ for some $\gamma > 0$ then

$$e^{-2\sum_{t=1}^{T}\gamma_t^2} \leq e^{-2\sum_{t=1}^{T}\gamma^2} = e^{-2T\gamma^2}$$

so that the training error approaches zero exponentially fast in the number of rounds T.

Boosting algorithms prior to AdaBoost satisfied similar bounds on the training error but required knowledge of γ, which can be hard to obtain. The beauty of AdaBoost is that it adapts naturally to the error rates of h_t so that knowledge of γ is not required (hence the name AdaBoost, short for Adaptive Boosting).

While the performance on the training data is reassuring, for a learning method to be valuable we are interested in the generalization error as opposed to the training error. The generalization performance of H, denoted $R(H)$, is given by

$$R(H) = P\{H(\bar{x}) \neq y\}$$

and is the probability that H misclassifies a new randomly drawn example.

Using VC theory, it can be shown that with high probability the error rate after T rounds of boosting is bounded as follows:

$$R(H) \leq \frac{1}{n}\sum_{i=1}^{n} I_{\{H(\bar{x}_i)\neq y_i\}} + O\left(\sqrt{\frac{TV}{n}}\right). \tag{17}$$

The problem with this bound is that it increases with the number of rounds T, suggesting that boosting will tend to overfit the training data as we run it for more rounds. However, while sometimes overfitting is observed, in many cases the generalization error continues to decrease as the number of rounds increases, and this can happen even after the error on the training data is zero. Explanations for this have been provided in terms of the margins of the classifier.

Here we use a slightly different way to measure the margin than was used in Section 7. Define the margin of the example $(\bar{x}_i, y_i)$ as

$$\text{margin}(\bar{x}_i, y_i) = \frac{y_i \sum_{t=1}^{T} \alpha_t h_t(\bar{x}_i)}{\sum_{t=1}^{T} \alpha_t} \tag{18}$$

The margin of $(\bar{x}_i, y_i)$ is always between -1 and $+1$ and is positive if and only if H classifies $(\bar{x}_i, y_i)$ correctly. In a sense, the margin measures the confidence in our classification of $(\bar{x}_i, y_i)$.

It can be shown that for any $\theta > 0$, with high probability we have

$$R(H) \leq \frac{1}{n} \sum_{i=1}^{n} I_{\{\text{margin}(\bar{x}_i, y_i) \leq \theta\}} + O\left(\sqrt{\frac{V}{n\theta^2}}\right). \tag{19}$$

An important feature of this bound is that it does not depend on T. This helps explain the empirical observation that running boosting for many rounds often does not increase the generalization error.

References

[1] M. Anthony and P.L. Bartlett, *Neural Network Learning: Theoretical Foundations*, Cambridge University Press, 1999.

[2] C. Bishop, *Pattern Recognition and Machine Learning*, Springer, New York, 2006.

[3] C.J.C. Burges, "A Tutorial on Support Vector Machines for Pattern Recognition," *Data Mining and Knowledge Discovery*, Vol. 2, pp.121167, 1998.

[4] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines*, Cambridge University Press, 2000.

[5] P.R. Devijver and J. Kittler, *Pattern Recognition: A Statistical Approach*, Prentice-Hall, 1982.

[6] L. Devroye, L. Györfi, G. Lugosi, *A Probabilistic Theory of Pattern Recognition*, Springer Verlag, New York, 1996.

[7] R.O. Duda, P.E. Hart, and D.G. Stork, *Pattern Classification*, Second Edition, Wiley, New York, 2001.

[8] K. Fukunaga, *Introduction to Statistical Pattern Recognition*, Second Edition, Academic Press, 1990.

[9] G. Harman and S.R. Kulkarni, *An Elementary Introduction to Statistical Learning Theory*, Wiley, 2011.

[10] G. Harman and S.R. Kulkarni, *Reliable Reasoning: Induction and Statistical Learning Theory*, MIT Press, 2007.

[11] T. Hastie, R. Tibshirani, J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Second Edition, Springer, 2009.

[12] S. Haykin, *Neural Networks: A Comprehensive Foundation*, Macmillan Publishing Company, 1994.

[13] M.J. Kearns and U.V. Vazirani, *An Introduction to Computational Learning Theory*, MIT Press, 1994.

[14] S.R. Kulkarni, G. Lugosi, and Venkatesh, S., "Learning Pattern Classification – A Survey," *IEEE Transactions on Information Theory*, Vol. 44, No. 6, pp. 2178-2206, Oct., 1998.

[15] D.J.C. MacKay, *Information Theory, Inference, & Learning Algorithms*, Cambridge University Press, 2002.

[16] T. Mitchell, *Machine learning*, McGraw-Hill, 1997.

[17] S. Russell and P. Norvig P, *Artificial Intelligence: A Modern Approach*, Third Edition, Prentice-Hall, 2011.

[18] R.J. Schalkoff, *Pattern Recognition: Statistical, Structural, and Neural Approaches*, Wiley, 1992.

[19] R.E. Schapire, "A brief introduction to boosting," *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, 1999.

[20] J. Shawe-Taylor and N. Cristianini, *Kernel Methods for Pattern Analysis*, Cambridge University Press, 2004.

[21] S. Theodoridis and K. Koutroumbas, *Pattern Recognition*, Fourth Edition, Academic Press, 2008.

[22] V.N. Vapnik, *The Nature of Statistical Learning Theory*, Springer-Verlag, New York, 1996.

[23] V. Vapnik, *Statistical Learning Theory*, Wiley-Interscience, New York, 1998.

[24] M. Vidyasagar, *A Theory of Learning and Generalization*, Springer-Verlag, 1997.